Wait-free Byzantine Agreement

Paul Attie
College of Computer Science
Northeastern University

Abstract

We study the problem of devising wait-free consensus protocols that tolerate Byzantine failures. We show that wait-free Byzantine agreement cannot be achieved even in the presence of a single faulty (i.e., Byzantine) process. We further show that wait-free weak Byzantine agreement cannot be achieved if shared objects are “resettable,” i.e., can be reset to an initial state from any of their reachable states by an appropriate sequence of operations. Resettable objects are preferable to non-resettable objects, since they can be reused more easily. We introduce a non-resettable object, a “sticky register,” which is a straightforward generalization of the sticky type of [Jay93], and use it devise a solution to wait-free weak Byzantine agreement that tolerates any number of faulty processes. Our impossibility results hold even if we can limit any process (including Byzantine processes) to invoking objects in a particular fixed set (i.e., impose a particular “access pattern”). Finally, we study the problem of repetitive wait-free weak Byzantine agreement: how to solve a sequence of successive instances of agreement. A straightforward solution uses an unbounded number of sticky registers—one for each instance of agreement. We show how to bound the number of registers needed by using failure detectors and weakening the wait-freedom property of the solution.

1 Introduction

Wait-freedom has been studied [Her91] extensively in recent years. A principal advantage of wait-free protocols is that the progress of each participating process is independent of the execution speeds of the other processes. In particular, a process can make progress even if the speed of all other processes is zero, i.e., if all other processes crash. Thus, a wait-free protocol with \( n \) participating processes can tolerate up to \( n - 1 \) crash failures. Little work has been done, however, on wait-free protocols that tolerate more severe types of faults, e.g., Byzantine faults. In this paper, we study the problem of devising wait-free shared memory consensus protocols that tolerate Byzantine faults. We show that wait-free Byzantine agreement cannot be achieved in the presence of only one Byzantine process. Our proof is straightforward, and exploits the validity condition for Byzantine agreement. We also show that wait-free weak Byzantine agreement can be achieved, but only by using “non-resettable” shared objects, that is, objects whose state cannot be reset to an initial state (by any sequence of permissible operations). With resettable objects, even a single Byzantine process is enough to prevent any solution to wait-free weak Byzantine agreement. Our impossibility results hold even if we can limit any process (including Byzantine processes) to invoking objects in a particular fixed set (i.e., impose a particular “access pattern”). Although this “limited access” model restricts the amount of damage a Byzantine process can do, the restriction is not powerful enough to enable a solution using resettable objects.

Using a non-resettable object (a sticky register — a generalization of Jayanti’s sticky type [Jay93] to more than two values) we give a protocol for wait-free weak Byzantine agreement. Non-resettable objects can serve for only one instance of agreement. To accommodate multiple instances (i.e., to solve the problem of repetitive Byzantine agreement) we need one sticky register for each instance of agreement. Because the amount of memory thus used grows unboundedly, this is not practical unless we can recover sticky registers that are “no longer needed.” To effect such recovery, we make the reasonable assumption that nonfaulty processes progress at some non-zero speed, and use a failure detector to exclude faulty processes from consideration. We can now recover all registers used for agreement instances that the slowest nonfaulty process has already participated in.

The rest of the paper is organized as follows. Section 2 defines our model of computation. Section 3 defines the wait-free Byzantine agreement problem and its weak variant. Sections 4 and 5 present our
impossibility results for wait-free Byzantine agreement and wait-free weak Byzantine agreement, respectively. Section 6 gives a solution to wait-free weak Byzantine agreement using a single sticky register. Section 7 studies the problem of repetitive wait-free weak Byzantine agreement, and Section 8 presents our conclusions and ideas for further work.

2 Model of Computation

A concurrent program \( P = (P_1 \parallel \cdots \parallel P_n, O_1, \ldots, O_m, A) \) consists of \( n \) sequential, nondeterministic processes that execute concurrently and interact by performing operations on the shared data objects \( O_1, \ldots, O_m. \) \( A \subseteq \{P_1, \ldots, P_n\} \times \{O_1, \ldots, O_m\} \) is an access pattern: \( P_i \) can perform operations on \( O_t \) only if \((P_i, O_t) \in A\). This access restriction remains in force even if \( P_i \) is Byzantine. Note that the access restriction is purely spatial: it only states which objects a process can invoke operations on. It says nothing about the temporal aspects of a process's behavior. An operation is a pair consisting of an invocation and the next matching response. A step of a process \( P_i \) is either an internal transition of \( P_i \), or an operation invocation by \( P_i \), or an operation response to \( P_i \). Processes and objects are asynchronous state-machines: there are no upper or lower bounds on the time it takes a process to execute a step or an object to respond to an invocation. Each process has a local state that is not visible to the other processes, and each shared object has a local state that is not visible to the processes except via the response of that object to invocations. We designate a subset of the local state space of each process (object) as the set of initial states of that process (object) respectively. We assume that all shared objects are reliable (i.e., they eventually respond to any invocation) and linearizable [HW90]. [MRL98] introduces the correctness criterion of Byzantine replaceability, which extends linearizability to a Byzantine environment. Our impossibility proofs also apply if this stronger correctness property of shared objects is assumed.

A global state is a tuple consisting of the local states of every process and shared object, and an initial global state is a tuple consisting of initial local states for every process and shared object. An execution is an alternating sequence of global states and process steps that starts in an initial global state. Hence, we model concurrency by nondeterministic interleaving. Also, we do not assume any form of fair scheduling. A maximal execution is an execution that is either infinite, or ends in a state in which no nonfaulty process can take a step. A reachable state is a state lying on some execution.

2.1 Notation and Technical Definitions

If \( \alpha \) is an execution, then \( \alpha \triangleright P_i \) is the result of removing from \( \alpha \) all steps of processes other than \( P_i \). Also, \( \alpha \triangleright O \) is the result of removing from \( \alpha \) all steps that are not invocations of operations on \( O \) or responses by \( O \). This extends to a set of objects in the obvious way: \( \alpha \triangleright (O_1, \ldots, O_t) \) results from removing all invocations of or responses by some object not in \( O_1, \ldots, O_t \). Finally, these notations can be nested, so that \( (\alpha \triangleright P) \triangleright O \) denotes the result of removing from \( \alpha \) all operations that are not invocations of \( O \) by \( P \) or responses to such invocations. We use \( O.\text{op}(\text{arglist}) \) to denote an operation \( \text{op} \) (with arguments \( \text{arglist} \)) on object \( O \), \( \text{invoke}(O.\text{op}(\text{arglist})) \) to denote an invocation of this operation, and \( \text{response}(O.\text{op}(\text{arglist})) \) to denote the matching response. If \( s \) is a global state, then \( s(P_i) \) is \( P_i \)'s local state in \( s \), and \( s(O) \) is object \( O \)'s local state in \( s \). If \( s \) and \( t \) are two global states, we say \( s \sim t \) iff \( s(P_i) = t(P_i) \) and \( s(O) = t(O) \) for all objects \( O \) that \( P_i \) has access to. For brevity, the constructions in this paper are expressed using pseudo-code. It is straightforward to translate this into a formal notation, e.g., I/O automata [LT88].

3 The Wait-free Byzantine Agreement problem

We propose the wait-free Byzantine agreement problem as follows. Each process starts with an initial value from a fixed set \( V \), and each nonfaulty processes eventually decides on some value in \( V \). The correctness conditions are as follows (the agreement and validity conditions are taken from [Lyn96, chapter 6]):

2
Agreement No two nonfaulty processes decide on different values.

Validity If all non-faulty processes start with the same initial value \( v \in V \), then \( v \) is the only possible decision value for a nonfaulty process.

Termination If \( P_i \) is nonfaulty, then \( P_i \) decides in every maximal execution.

No-conditional-waiting If \( P_i \) is a nonfaulty process and \( s \) is a reachable state in which \( P_i \) has not decided, then there exists an execution \( \alpha \) starting in \( s \) such that:

1. \( \alpha \) consists only of steps of \( P_i \).
2. \( P_i \) decides in \( \alpha \).

Uniform-initial-state Every combination of initial values for each process and initial local state for each shared object is a possible initial global state (of any solution).

The Termination condition rules out busy waiting, and the No-conditional-waiting condition rules out conditional waiting. Together, these impose the wait-freedom requirement (cf. the definition in [Her91, p. 129]). We also define the wait-free weak Byzantine agreement problem, in which the Validity condition is replaced by the Weak Validity condition:

Weak Validity If there are no faulty processes and all processes start with the same initial value \( v \in V \), then \( v \) is the only possible decision value.

We impose the uniform initial state condition in order to rule out trivial solutions in which each process has a "shared" object that only it can access, and whose initial value gives the correct decision value. In the sequel, we assume without loss of generality that \( \{0, 1\} \subseteq V \).

4 Impossibility of Wait-free Byzantine Agreement

In this section, we show that there is no solution to wait-free Byzantine agreement, even if a single process exhibits Byzantine behavior. Our first proposition shows that, when a process \( P_i \) runs alone and decides before other processes take any steps, then it must decide its own initial value. Intuitively, this follows because \( P_i \)'s decision must be independent of the initial values of the other processes, and so, to satisfy validity, \( P_i \) must decide on its own initial value.

Proposition 1 Let \( P = (P_1 \parallel \cdots \parallel P_n, O_1, \ldots, O_m, A) \) be a protocol for wait-free Byzantine agreement. In any execution of \( P \) in which \( P_i \), \( i \in \{1..n\} \), runs alone and decides before any other process takes a step, \( P_i \) must decide its own initial value.

Proof. Let \( s \) be an arbitrary global initial state, and let \( \alpha_s \) be an arbitrary execution starting in \( s \) in which \( P_i \) first runs alone until it decides. Let the initial value of \( P_i \) in \( s \) be \( v_i \). Let \( t \) be an arbitrary global initial state which is identical to \( s \) except that all processes have initial value \( v_i \) (thus all shared objects have the same initial local states in \( s \) as in \( t \)). Starting in state \( t \), it is possible for \( P_i \) to execute the same steps that it executes in \( \alpha_s \), since \( P_i \)'s local state and all object states are the same in \( s \) as in \( t \). Let \( \alpha \) be the resulting execution. By validity, \( P_i \) decides \( v_i \) in \( \alpha \). Hence, \( P_i \) also decides \( v_i \) in \( \alpha_s \) since it executes the same steps in both executions. Since \( \alpha_s \) was chosen arbitrarily, the proposition follows. \( \Box \)

We first establish the impossibility result for three processes. The generalization to \( n \) processes is easily done using the no-conditional-waiting and uniform-initial-state requirements.

Theorem 2 There is no solution to the wait-free Byzantine agreement problem for three processes if one of them can be faulty.
Proof. By contradiction. Suppose a solution \((P_1 \parallel P_2 \parallel P_3, O_1, \ldots, O_m, A)\) exists. Let \(\alpha_1\) be the following execution of this solution: \(P_1, P_2\) are nonfaulty and have initial values 1, 0 respectively. First, \(P_1\) runs alone until it decides. By proposition 1, \(P_1\) decides 1. Then, \(P_2\) runs alone until it decides. By the agreement condition, \(P_2\) also decides 1.

Now consider execution \(\alpha_2\) which is as follows. \(P_1\) is faulty, \(P_2\) is nonfaulty and has the same initial state as in \(\alpha_1\) (and therefore has initial value 0), and \(P_3\) is nonfaulty and has initial value 0. Also, the shared objects have the same initial states as in \(\alpha_1\). First, \(P_1\) runs alone, and executes exactly the same steps as it does in \(\alpha_1\). This is possible since the shared objects have the same initial values and \(P_1\) is faulty (and can therefore deviate from the protocol). Next, \(P_2\) runs alone until it decides. Since the shared objects have the same values in \(\alpha_1\) and \(\alpha_2\) when \(P_2\) starts executing, and \(P_2\) has the same initial state, it is possible for \(P_2\) to execute exactly the same sequence of steps that it executed in \(\alpha_1\). Let this then be the sequence of steps that \(P_2\) executes in \(\alpha_2\). Hence \(P_2\) decides 1 in \(\alpha_2\), since it decides 1 in \(\alpha_1\). But this contradicts the validity condition, and we are done. \(\square\)

Theorem 3 There is no solution to the wait-free Byzantine agreement problem for \(n \geq 2\) processes if one of them can be faulty.

Proof. For \(n = 2\), the proof is essentially the same as that for Theorem 6.27 in [Lyn96], and is omitted. For \(n \geq 3\), we assume a solution exists and derive a contradiction exactly as in the proof of Theorem 2. From the wait-freedom condition and the uniform initial state condition, it is clear that the executions \(\alpha_1, \alpha_2\) (with \(\alpha_1\)'s initial state extended arbitrarily to \(P_4, \ldots, P_n\), and \(\alpha_2\)'s initial state chosen so that \(P_4, \ldots, P_n\) also have initial value 0) constructed in the aforementioned proof are executions of a solution for any \(n \geq 3\). \(\square\)

5 Impossibility of Wait-free Weak Byzantine Agreement using Resettable Objects

While wait-free Byzantine agreement is not achievable, weak wait-free Byzantine agreement can be solved, but only by using shared objects which cannot be reset to an initial state. We define:

Definition 1 An object is resettable if, for each of its reachable states, there exists a sequence of operations that takes the object back to some initial state. We call such a sequence of operations a reset sequence.

Note that our definition allows an object to be reset to several initial states from a given reachable state \(s\). The object cannot however, be reset to an initial state that is not reachable from \(s\). An extreme case is when every initial state is reachable from \(s\), and so the object could be reset to any initial state (from \(s\)).

Resettable objects are more desirable than non-resettable objects, because they can be easily reused (e.g., for successive instances of agreement). A non-resettable object can only be “reused” by reclaiming the memory that it occupies, i.e., destroying it, and dynamically creating a new instance of the same object type. We examine these issues further in Section 7.

In this section, we show that there is no solution to wait-free weak Byzantine agreement using only resettable objects, and in the presence of a single Byzantine process. In the next section, we exhibit a solution that uses a non-resettable object: a sticky register.

Proposition 4 Let \(P = (P_1 \parallel \cdots \parallel P_n, O_1, \ldots, O_m, A)\) be a protocol for wait-free weak Byzantine agreement. In any execution of \(P\) in which \(P_i\), \(i \in \{1,n\}\), runs alone and decides before any other process takes a single step, \(P_i\) must decide its own initial value.

Proof. In the proof of Proposition 1, no processes are assumed faulty. Hence, the proof still applies if we assume that all processes are nonfaulty, and we replace the use of validity by the use of weak validity. \(\square\)

As before, we present the impossibility result for three processes, and easily generalize it to \(n\) processes using the no-conditional-waiting and uniform-initial-state requirements.
Theorem 5 There is no solution to the wait-free weak Byzantine agreement problem for three processes if one of them can be faulty and all shared objects are resettable.

Proof. By contradiction. Suppose a solution $P = (P_1 \parallel P_2 \parallel P_3, O_1, \ldots, O_m, A)$ exists. Let $O_{ij} \subseteq \{O_1, \ldots, O_m\}$ be the set of objects that can be accessed by $P_i$ and $P_j$, $i, j \in \{1, 2, 3\}, i \neq j$, and $O_{123} \subseteq \{O_1, \ldots, O_m\}$ be the set of objects that can be accessed by $P_1, P_2$, and $P_3$, all according to the access pattern $A$. We construct two executions $\alpha_1$ and $\alpha_2$ of $P$ as follows (Figure 1 shows how $\alpha_1$ and $\alpha_2$ are "alternately" constructed from each other in a well-defined manner).

The first part of $\alpha_1$ is as follows. $P_1$ is faulty, $P_3$ has initial value 1, and $O_{123}$ is in some initial state $\mathcal{V}_{23}$.

First, $P_2$ runs alone until it decides. Without loss of generality, suppose $P_2$ decides 1. Next, $P_1$ invokes reset sequences for all objects in $O_{13}$ and $O_{123}$. Let the resulting initial states of $O_{13}, O_{123}$ be $\mathcal{V}_{13}, \mathcal{V}_{123}$ respectively.

$\alpha_2$ is as follows. $P_2$ is faulty, $P_1$ is nonfaulty and has initial value 0, and $P_3$ is nonfaulty and has the same initial state as in $\alpha_1$ (and therefore has initial value 1). The initial states of $O_{13}, O_{23}, O_{123}$ are $\mathcal{V}_{13}, \mathcal{V}_{23}, \mathcal{V}_{123}$ respectively (by the uniform-initial-state requirement, these initial conditions must exist for some execution). First, $P_1$ runs alone until it decides. By Proposition 4, $P_1$ decides 0. Next, $P_2$ executes $(\alpha_1 \uparrow P_2) \uparrow O_{23}$ (since $P_2$ is faulty, it can execute any sequence of operations on objects it has access to). Let the global state at this point be $\alpha_2,t$. Finally, $P_3$ runs alone until it decides. By agreement, $P_3$ must decide 0, since $P_1$ decided 0, and $P_1, P_3$ are nonfaulty.

The last part of $\alpha_1$ is as follows. $P_1$ executes $(\alpha_2 \uparrow P_1) \uparrow (O_{13} \cup O_{123})$—since $P_1$ is faulty, it can execute any sequence of operations on objects it has access to. Let the global state at this point be $\alpha_1.s$. Finally, $P_3$ runs alone until it decides. By agreement, $P_3$ must decide 1, since $P_2$ decided 1, and $P_2, P_3$ are nonfaulty.

We now establish $\alpha_1.s \not\simeq \alpha_2,t$. In $\alpha_1.s$, the state of $O_{23}$ results from the sequence $(\alpha_1 \uparrow P_2) \uparrow O_{23}$ applied to the initial state $\mathcal{V}_{23}$ of $O_{23}$ (by definition of $\uparrow$). In $\alpha_2,t$, the state of $O_{23}$ results from the same sequence, namely $(\alpha_1 \uparrow P_2) \uparrow O_{23}$, applied to the same initial state $\mathcal{V}_{23}$ (by construction of $\alpha_2$). Hence $\alpha_1.s(O_{23}) = \alpha_2,t(O_{23})$.

In $\alpha_1.s$, the states of $O_{13}, O_{123}$ result from the sequence $(\alpha_2 \uparrow P_1) \uparrow (O_{13} \cup O_{123})$ applied to the states $\mathcal{V}_{13}, \mathcal{V}_{123}$ respectively (because of the reset sequences invoked by $P_1$ in $\alpha_1$, the actual initial states of $O_{13}, O_{123}$ don’t matter). In $\alpha_2,t$, the states of $O_{13}, O_{123}$ also result from the sequence $(\alpha_2 \uparrow P_1) \uparrow (O_{13} \cup O_{123})$ applied to the (initial) states $\mathcal{V}_{13}, \mathcal{V}_{123}$ respectively (by definition of $\uparrow$). Hence we conclude $\alpha_1.s(O_{13}) = \alpha_2,t(O_{13}), \alpha_1.s(O_{123}) = \alpha_2,t(O_{123})$. Finally, the local state of $P_3$ in $\alpha_1.s$ is its initial state in $\alpha_1$, since $P_3$ takes no steps in the prefix of $\alpha_1$ up to $\alpha_1.s$. Likewise the local state of $P_3$ in $\alpha_2,t$ is its initial state in $\alpha_2$. But these initial states are the same, by construction of $\alpha_2$. Hence $\alpha_1.s(P_3) = \alpha_2,t(P_3)$. Since $P_3$’s state and the states of all objects that $P_3$ can access are the same in $\alpha_1.s$ as in $\alpha_2,t$, we conclude $\alpha_1.s \not\simeq \alpha_2,t$.

In $\alpha_1$, $P_3$ runs alone from $\alpha_1.s$ until it decides. In $\alpha_2$, $P_3$ runs alone from $\alpha_2,t$ until it decides. Since $\alpha_1.s \not\simeq \alpha_2,t$, and $P_3$ runs alone from $\alpha_1.s$, $\alpha_2,t$ until it decides, it is possible for $P_3$ to execute the same sequence (i.e., $\alpha_1 \uparrow P_3$) of steps in $\alpha_2$ as in $\alpha_1$. Let this then be the sequence that $P_3$ actually executes in $\alpha_2$. Hence, $P_3$ must decide the same value in $\alpha_2$ as in $\alpha_1$. But we showed above that $P_3$ decides 1 in $\alpha_1$ and 0 in $\alpha_2$. Hence the desired contradiction. \hfill \Box

Theorem 6 There is no solution to the wait-free weak Byzantine agreement problem for $n \geq 3$ processes if one of them can be faulty and all shared objects are resettable.

Proof. We assume a solution exists and derive a contradiction exactly as in the proof of Theorem 5. From the wait-freedom condition and the uniform initial state condition, it is clear that the executions $\alpha_1, \alpha_2$ (with their initial states extended arbitrarily to $P_4, \ldots, P_n$) constructed in the aforementioned proof are executions of a solution for any $n \geq 3$.

We note that if an access pattern is not imposed, then the proof of Theorem 5 becomes trivial: let $P_1$ be nonfaulty and run alone until it decides; then let $P_2$ be faulty and reset all objects in the system; finally

\footnote{For brevity, we discuss the sets of objects $O_{ij}, O_{123}$ as if they were single objects. For example, $\mathcal{V}_{23}$ is actually a set of initial object states.}
let $P_3$ be nonfaulty and run alone until it decides. Invoking Proposition 4 then allows us to conclude that $P_1$ and $P_3$ must both decide on their initial values, since $P_3$’s view of the execution is indistinguishable from its view of some execution in which it runs alone and decide before any other process takes a step.

6 Wait-free Weak Byzantine Agreement Using One Sticky Register

A sticky register is a register whose value can be set once, and subsequently can never be changed. It is essentially the same as the consensus type of [Her91], and generalizes the sticky type of [Jay93] from a binary domain to an arbitrary domain (cf. also the sticky bit of [Pl89]). A sticky register $r$ has initial value $\perp \in V^2$, and supports a single operation, $r.\text{jam}(v)^3$, that sets $r$’s value to $v$ if the value is $\perp$ and otherwise leaves $r$ unchanged. In either case, $r.\text{jam}(v)$ returns the resulting value of $r$. The formal definition of $r.\text{jam}(v)$ is as follows:

$r.\text{jam}(v : V)$ returns($V$)

if $r = \perp \land v \in V$ then $r := v$ endif;
return($r$)

Given a sticky register $r$, we have a trivial protocol $(P_1 \parallel \cdots \parallel P_n, r)$ that solves wait-free weak Byzantine agreement: each $P_i$ executes $r.\text{jam}(v_i)$, where $v_i$ is $P_i$’s initial value, and decides on the value returned. The protocol for $P_i$, $i \in \{1..n\}$, is:

$\text{agree}(r : \text{sticky register}, v_i : V)$

invoke($r.\text{jam}(v_i)$);
$\hat{d}_i := \text{response}(r.\text{jam}(v_i))$;
$\text{decide}(d_i)$

**Proposition 7** The above protocol solves the wait-free weak Byzantine agreement problem, in the presence of any number of Byzantine processes.

---

3Recall that $V$ is the domain of possible decision values.

3We borrow Plotkin’s [Pl89] terminology here.
Proof. For agreement, let $v$ be the value returned by the first $\text{jam}$ operation. By definition of $\text{jam}$, all subsequent $\text{jam}$ operations also return $v$, even in the presence of Byzantine behavior by any number of processes. Hence all nonfaulty processes decide $v$. For weak validity, if all processes are nonfaulty and have the same initial value $v$, then the first $\text{jam}$ operation must be $r.\text{jam}(v)$, by definition of the protocol. Hence, all $\text{jam}$ operations return $v$, and so all processes decide $v$. Termination and no-conditional-waiting are obvious, since nonfaulty processes execute a finite amount of straight-line code that contains no tests. Finally, the sticky register $\sigma$ has only one possible initial state, namely $\perp$, and the only local state that each process has is its initial value. Hence it is clear that the protocol permits all possible initial global states, i.e., the uniform initial state requirement is satisfied.

It is easy to see that this protocol works because the behavior of a sticky register severely limits the amount of damage that a Byzantine process can inflict. All that a Byzantine process can do is set a sticky register to an arbitrary value in $V$. This is sufficient to destroy validity (and hence Theorem 3), but not to destroy agreement.

7 Repetitive Wait-free Weak Byzantine Agreement

The protocol of Section 6 uses a single sticky register to solve a single instance of wait-free weak Byzantine agreement. Thus, to solve repetitive agreement—where processes repeatedly engage in successive instances of agreement—we need an unbounded number of sticky registers. This is because sticky registers are not settable, and so cannot be reused, which means that we need a new sticky register for each instance of agreement. We assume that an underlying, reliable operating system is responsible for allocating and deallocating registers. We represent the operating system as an additional process $OS$.

Our protocol for repetitive agreement is as follows. We number the instances of agreement with the positive integers. We also number each sticky register with the number of the agreement instance that the register is used for. Each process $P_i$ maintains an instance variable $\text{instance}_i$. After participating in some instance of agreement, $P_i$ increases $\text{instance}_i$ to the number of the next agreement instance that it wishes to participate in. Note that the increments of $\text{instance}_i$ need not be 1, i.e., not every nonfaulty process need participate in every agreement instance.

As mentioned above, this protocol uses an unbounded number of sticky registers. To bound the number of sticky registers needed at any time, we:

1. Reclaim registers that are “no longer needed.” A register is no longer needed when the “slowest” nonfaulty process has participated in the agreement instance that the register is used for.

2. Occasionally force a “fast” nonfaulty process to wait for more sticky registers to be allocated. This entails giving up some aspects of wait-freedom. Specifically, processes are still wait-free “relative to one another,” i.e., one process cannot delay another, but processes are not wait-free relative to the operating system ($OS$), since $OS$ can delay a process by forcing it to wait for a sticky register to be allocated. We define pseudo-wait-free Byzantine agreement by modifying the termination and no-conditional-waiting requirements as follows:

- **Termination** If $P_i$ is nonfaulty, then $P_i$ decides in every maximal execution in which $OS$ allocates the required sticky register.

- **No-conditional-waiting** Let $P_i$ be a nonfaulty process and $s$ be a reachable state in which $P_i$ has not decided. Then, if there exists an execution starting in $s$ in which $OS$ allocates the required sticky register, then there also exists an execution $\alpha$ starting in $s$ such that:
  (a) $\alpha$ consists only of steps of $P_i$ or $OS$.
  (b) $P_i$ decides in $\alpha$.

By using memory freed up by reclamation to allocate new registers, we can bound the total amount of memory used for sticky registers at any time. Also, since processes probably have to wait for the operating
system at some point, the replacement of wait-freedom by pseudo-wait-freedom is unlikely to be significant from a practical point of view. The above discussion suggests a generalization of wait-freedom to “wait-freedom relative to another particular process.” We defer the formal definition and exploration of this notion to another occasion.

To reclaim registers that are no longer needed, OS periodically executes:

\[ min_{\text{inst}} := \min_{\text{nonfaulty instance}} \]

and recovers all sticky registers up to register number \( min_{\text{inst}} - 1 \). The main problem then, is detecting faulty processes and excluding them from consideration when computing \( min_{\text{inst}} \). Unfortunately, since one of the possible behaviors of a Byzantine process is to behave “correctly,” it is not possible to detect all faulty processes. Following [MR97], we detect only those Byzantine processes whose behavior prevents progress, where in this case progress occurs when \( min_{\text{inst}} \) is increased, thereby allowing some more sticky registers to be reclaimed. Thus, for purposes of this section only, we redefine a faulty process as follows:\(^4\):

\[ P_i \text{ is faulty iff } \exists v, \Box (instance_i^{OS} < v) \lor \neg \forall v, \Box (instance_i^{OS} = v) \Rightarrow \Box (instance_i^{OS} \geq v). \]

Here, \( instance_i^{OS} \) is a variable local to the operating system process OS which records OS’s most recent observation of the value of \( instance_i \). Since our main concern here is that faulty processes prevent the reclamation of registers, it suffices to exclude from consideration faulty processes that appear to be slow to OS. Faulty processes that OS “perceives” to be fast, i.e., whose \( instance_i \) variables are always increasing when sampled by OS, will not prevent register reclamation. For purposes of register reclamation, these processes can be treated by OS as if they were correct.

Our definition of faulty above considers a process faulty iff its observed instance number is either bounded by some value \( v \), or decreases at some point. In the first case, this prevents reclamation of registers numbered higher than \( v \). In the second case, if the observed instance numbers of two or more processes alternately show decreases in such a way that their minimum is bounded by some value \( v \) (i.e., the decreases “overlap”) this would again prevent reclamation of registers numbered higher than \( v \). To detect faulty processes, we employ a failure detector \( D \) [CT96]. Unlike the definition of failure detector given in [CT96], where each process has a failure detector module, our failure detector consists of only one module, which is accessible to OS. \( D \) uses the instance numbers \( instance_i^{OS} \) which OS observes. The completeness and strong accuracy properties of failure detectors given in [CT96] are now rephrased as follows:

**Completeness** Eventually, every process that fails is permanently suspected by OS.\(^5\)

**Strong accuracy** No process is suspected by OS before it fails.

**Eventual strong accuracy** There is a time after which correct processes are not suspected by OS.

Figure 2 gives our solution for repetitive pseudo-wait-free weak Byzantine agreement with register reclamation. Each \( P_i \) repeatedly participates in the instance of agreement given by its \( instance_i \) variable, and then sets \( instance_i \) to the next value of a monotonically increasing sequence \( schedule_i \), which gives the instances of agreement that \( P_i \) participates in.\(^6\) We assume that \( v_{instance_i} \) gives the initial value of \( P_i \) for the agreement instance identified by \( instance_i \).

We require that \( D \) be such that the following hold:

\[ \neg \exists v, \Box (min_{\text{inst}} < v) \quad (1) \]

\[ \forall \text{ nonfaulty } P_i, \Box (min_{\text{inst}} \leq instance_i) \quad (2) \]

\(^4\)Note that we use first-order linear-time temporal logic [Eme90]. \( \Box, \Diamond \) are the linear temporal logic operators “always” and “eventually,” respectively. Thus, \( \Box \Diamond f \) means that there exists a state after which \( f \) holds continuously.

\(^5\)Since there is only one failure detector module, the strong and weak completeness properties of [CT96] coincide.

\(^6\)This “infinite” representation is for convenience only. In practice, \( P_i \) would compute “on the fly” the number of the next instance of agreement that it wishes to participate in.
\( P_i : \)
\[ k_i := 1; \]
\[ \text{instance}_i := \text{schedule}_i[k_i]; \]
\begin{verbatim}
repeat forever
  wait until OS has allocated registers up to register number \text{instance}_i;
  agree(r_{instance_i}, v_{instance_i});
  k_i := k_i + 1;
  \text{instance}_i := \text{schedule}_i[k_i]
endrepeat
\end{verbatim}

\( OS :: \) repeat forever
\begin{verbatim}
  forall \( i \in \{1..n\} \) instance_i^{OS} := \text{instance}_i endfor;
  min\_inst := \text{MIN}_{P \notin D} instance_i^{OS};
  reclaim all registers up to register min\_inst - 1;
  allocate some new registers, possibly using memory freed up by reclamation
endrepeat
\end{verbatim}

Figure 2: Repetitive pseudo-wait-free weak Byzantine agreement with register reclamation.

Formula (1) requires that \( \text{min\_inst} \) increase without bound. This enables the continuing reclamation of sticky registers. (2) requires that \( \text{min\_inst} \) is never greater than the instance number of a nonfaulty process. This ensures that no register is reclaimed before it has been used by every nonfaulty process that needs it (i.e., every nonfaulty process that participates in the agreement instance which the register implements).

We show that completeness and strong accuracy of \( D \) are sufficient to assure (1) and (2).

**Proposition 8** If \( D \) satisfies completeness and strong accuracy, then (1) and (2) hold.

**Proof.** By completeness of \( D \), every faulty process is eventually excluded from the computation of \( \text{min\_inst} \). Since \( \text{instance}_i^{OS} \) increases without bound when \( P_i \) is nonfaulty, it follows that \( \text{min\_inst} \) also increases without bound, and so (1) holds. By strong accuracy of \( D \), \( \text{instance}_i^{OS} \) is included in the computation of \( \text{min\_inst} \) for every nonfaulty \( P_i \), and so \( \text{min\_inst} \leq \text{instance}_i^{OS} \). Since \( \text{instance}_i \) increases monotonically, it follows that \( \text{instance}_i^{OS} \leq \text{instance}_i \), is invariant (assume that \( \text{instance}_i^{OS} \) is initialized to, say, 0). Hence \( \text{min\_inst} \leq \text{instance}_i \) is invariant, which establishes (2).

Strong accuracy may be difficult to achieve in practice. Suppose we use a failure detector that satisfies completeness and eventual strong accuracy. It is easy to see that (1) continues to hold, but that (2) may not, because now a nonfaulty process \( P_i \) may be incorrectly suspected, which would cause \( \text{instance}_i \) to be excluded from the calculation of \( \text{min\_inst} \). However, the following does hold:

\[ \forall \text{nonfaulty } P_i, \Box (\text{min\_inst} \leq \text{instance}_i) \quad (2') \]

In other words, (2) holds from some point onwards. Hence, replacing a strongly accurate detector by an eventual strongly accurate detector results in a finite number of agreement instances not executing correctly because the needed sticky register was reclaimed too soon (the solution of Figure 2 must be modified so that invocations of agree(r_{instance_i}, v_{instance_i}) return a default value if r_{instance_i} has been reclaimed—this still satisfies agreement, but weak validity may be violated).

**Proposition 9** If \( D \) satisfies completeness and eventual strong accuracy, then (1) and (2') hold.

**Proof.** The proof of (1) is exactly the same as in the proof of Proposition 8 above. By eventual strong accuracy of \( D \), \( \text{instance}_i^{OS} \) is eventually included in every computation of \( \text{min\_inst} \) for every nonfaulty \( P_i \) (from some point on), and so \( \Box (\text{min\_inst} \leq \text{instance}_i^{OS}) \). Since \( \text{instance}_i^{OS} \leq \text{instance}_i \), is invariant, \( \Box (\text{min\_inst} \leq \text{instance}_i) \) holds, which establishes (2').

9
8 Conclusions and Further Work

In this paper, we studied the problem of wait-free consensus in the presence of Byzantine failures. We showed that wait-free Byzantine agreement is not achievable, and that weak wait-free Byzantine agreement is achievable only if we use non-resettable objects. Our impossibility results hold in the presence of only a single Byzantine process, and even if we can impose an “access pattern” that, for each process, fixes the objects that the process has access to. We also gave a protocol for weak wait-free Byzantine agreement that uses non-resettable objects, and tolerates any number of Byzantine faults. Our results suggest that the usual “majority voting” considerations that result in the usual bound of at most one-third faulty processes \( n \geq 3f + 1 \) do not come into play when wait-freedom is required. Finally, we studied repetitive weak wait-free Byzantine agreement, and showed that it is possible to bound the amount of memory needed by using failure detectors and weakening the wait-freedom property of the solution.

The access patterns considered in this paper either grant a process access to all of the operations defined by the type of a particular shared object, or to none of them. A more refined notion would grant different processes access to different subsets of the defined operations. Thus, some processes may only have access to a “read” operations. It would be worthwhile to investigate whether our impossibility results carry over to this more refined model. Also, the notion of wait-freedom relative to another process is worth investigating further. Finally, our solution of repetitive wait-free weak Byzantine agreement should be extended to use bounded instance numbers.

References


